Exploring ChatGPT Efficiency in Automatic Test Generation for Python: A Comparative Analysis

Lucca Renato Guerino
Departmento de Computação
Universidade Federal de São Carlos
São Carlos, SP, Brazil
luccaguerino@estudante.ufscar.br

Auri Marcelo Rizzo Vincenzi*
Faculdade de Engenharia
Universidade do Porto
Porto, Portugal
auri@fe.up.pt

ABSTRACT

Context: Large language models (LLMs) like ChatGPT have gained attention in automated software testing. This study evaluates ChatGPT-3.5-turbo's ability to generate test sets for Python programs, comparing it with Pynguin and pre-existing test sets. **Problem**: Automated testing remains challenging for dynamically typed languages like Python, requiring adaptable tools for diverse code structures. **Solution**: We assessed ChatGPT-3.5-turbo's test generation using different prompt configurations and temperature settings. **Method**: Using 40 Python programs, we generated Pytest-compliant tests via the OpenAI API, varying temperature settings (0.0 to 1.0). Tests were validated using Pytest, with coverage and mutation scores measured via Coverage, MutPy, and Cosmic-Ray. Pynguin-generated and pre-existing test sets served as baselines. **Summary of Results**: ChatGPT-3.5-turbo successfully generated valid tests for simpler programs, but averaged below 28% overall, with a low cost. Higher temperatures (0.5–1.0) improved results, but combining test cases from all temperatures introduces diversity in the LLM-generated test sets, making it possible to overcome both Pynguin and pre-existing test sets in terms of decision coverage and mutation score.

KEYWORDS

software testing, experimental software engineering, automated test generation, large language models, coverage testing, mutation Testing, testing tool

1 Introduction

In recent years, large language models (LLMs) have emerged as transformative tools across various technological domains. Among these, OpenAI's ChatGPT[1] series has gained traction due to its versatile applications in natural language processing and conversational AI. ChatGPT has been widely adopted in professional and academic settings, despite its costs, and has been featured in numerous recent works. This rapid adoption has sparked interest in evaluating its potential to support and enhance critical software engineering tasks, particularly in software testing [13].

With increasing system complexity and the demand for agile development, automating testing is crucial for ensuring software quality. Despite the use of automated tools, challenges persist, especially in dynamically typed languages like Python, where diverse code structures necessitate adaptive testing strategies. This study examines whether LLMs, specifically ChatGPT-3.5-turbo, can address these challenges and support rigorous software testing processes.

Python's prominence as the most popular programming language, according to the Tiobe Index [14], makes it an ideal candidate for investigation. Its versatility and adoption across industries ensure broad applicability. This study explores the potential of LLM (ChatGPT) to enhance software testing processes. Automated test generation with LLMs promises improved efficiency, scalability, and reliability, which in turn impact the development and maintenance of robust systems.

The generated test suites are evaluated using Coverage[2] for decision coverage and MutPy [8] and Cosmic-Ray [3] for mutation testing. Mutation testing assesses the robustness of a test suite against potential faults, ensuring its reliability. Pre-existing test sets serve as benchmarks to evaluate LLM-generated tests.

The document is structured as follows: Section 2 provides background information. Section 3 reviews related works. Section 4 details the experiment design, including prompts and test parameters. Section 5 describes data collection and analysis. Section 6 discusses findings, while Section 7 addresses limitations. Finally, Section 8 presents conclusions and future research directions.

2 Background

This section outlines essential software testing and related concepts to facilitate understanding of the remainder of the paper.

Automated test generation, which uses algorithms and tools to create test scenarios without manual input, enhances coverage by systematically exploring inputs and program paths. Automating test creation aims to improve fault detection and software quality while reducing manual effort.

LLMs have shown promise in automated test generation, aiding unit testing, debugging, and program repair. Wang et al. [17] demonstrate their potential to streamline testing and improve software quality. However, challenges remain in achieving comprehensive coverage, handling complex test oracles, and rigorously evaluating LLM performance in real-world applications. Python's dynamic typing system complicates automated test generation [16], as variables lack fixed types and may change during execution [5, 9]. Identifying patterns for test generation is particularly difficult in this context.

Pynguin represents a significant advancement in Python test generation [10], leveraging mutation-based techniques and evolutionary algorithms to enhance unit test effectiveness. By interpreting Python bytecode, Pynguin improves code coverage and reliability. The decision to use Pynguin in our study is based on the work of Guerino and Vincenzi [7], which identifies it as the

[1] https://chat.openai.com/

[2] https://pypi.org/project/coverage/

most suitable automated testing tool for Python, operating independently without human intervention and relying solely on the program under test.

Software testing ensures applications meet specified requirements and function correctly. Mutation testing, a key technique, evaluates test suite effectiveness by introducing minor code modifications (mutants) and assessing whether the tests detect them [4]. Mutation score measures effectiveness as the ratio of killed mutants to total non-equivalent mutants [4]. Mutants that are detected and cause the test suite to fail are called 'killed mutants," while those that remain undetected and pass the test suite are known as live mutants.". A higher score indicates a stronger test suite, improving fault detection [1, 11].

Advancements in mutation testing automation have led to the development of optimized tools. This study focuses on MutPy[3] and Cosmic-Ray[4], identified by other researchers as the two most adequate mutation testing tools for Python [6]. They automate the generation of mutants and test execution, enhancing efficiency and accuracy. Key differences include mutation operators, mutant volume, and data processing. Operator diversity affects test depth, excessive mutants increase processing time, and result interpretation impacts mutation testing.

3 Related Work

In their work, Wang et al. [17] present a comprehensive survey of 102 studies on LLMs in software testing, covering applications such as test case preparation, debugging, and program repair. They highlight test generation and bug repair as primary uses, describe prompt engineering and techniques like mutation and differential testing to improve performance, and outline challenges including coverage gaps and the oracle problem. The survey also suggests future directions in early-stage testing and advanced prompt design, offering a roadmap for LLM-driven progress in software testing.

Guerino et al. [6] conducted a comparative study on Python mutation testing tools, specifically Cosmic-Ray, MutPy, MutMut, and Mutatest, assessing them from static and dynamic perspectives. The study introduces an evaluation framework adapted from Java tools, used for a static assessment of documentation and features, as well as a dynamic analysis with cross-tool testing to measure each tool's efficacy in mutant detection. The results indicate Cosmic-Ray offers the most customization, while MutPy shows superior mutant detection capabilities, but both overcome MutMut and Mutatest.

Zhang et al. [18] perform an extensive study on LLMs for automated assertion generation, addressing the challenges of manually writing assertions in unit tests. They evaluate the accuracy of five LLMs against six techniques across two datasets, demonstrating that LLMs outperform state-of-the-art approaches, particularly CodeT5, which achieves significant gains in assertion accuracy. They also assess the bug-detection capabilities of LLM-generated assertions and propose a novel retrieval-and-repair approach, which further enhances prediction accuracy. Their findings highlight the potential of LLMs in reducing manual testing efforts in software engineering.

The paper by Sallou et al. [12] critically examines the risks of using LLMs in software engineering, focusing on threats to validity from closed-source dependencies, data leakage, and reproducibility issues. They highlight how LLM pre-training on widely available datasets can lead to overfitting on evaluation tasks, such as test generation, and recommend methodologies for increasing transparency and ensuring rigorous assessment, such as code obfuscation and metamorphic testing, to mitigate potential biases and improve the robustness of LLM applications in software engineering.

Considering the classification provided by Wang et al. [17], our research focuses on unit test generation, evaluating ChatGPT's effectiveness in zero-shot and few-shot contexts. Our primary contribution is the comparison of the effectiveness of the generated tests via ChatGPT, Pynguin, and pre-existing ones against two well-known testing criteria: decision coverage and mutation testing. We also discuss cost aspects regarding the number of generated tests and tokens used during ChatGPT interactions. Finally, we assess LLM-generated tests' ability to detect specific faults using mutation operators from MutPy and Cosmic-Ray.

4 Experiment Design

The experiment design of this paper is guided by the Goal-Question-Metric (GQM) methodology, with each GQM section detailed below.

Goal

Improve and evaluate the effectiveness, fault detection capability, and cost-efficiency of using ChatGPT-3.5-turbo for automated test generation in Python, comparing its performance to Pynguin and pre-existing test sets. The goals encompass test validity, coverage, mutation score, and resource consumption.

Questions

RQ1. How capable is ChatGPT-3.5-turbo at autonomously generating valid test cases for Python modules using its API?
RQ2. How does the test adequacy (coverage, mutation score) of ChatGPT-3.5-turbo compare to an established automated Python test generation tool (Pynguin) and pre-existing test sets?
RQ3. To what extent do LLM-generated test sets detect different fault categories, as represented by the mutation operators of MutPy and Cosmic-Ray?

Metrics

- For RQ1 (Capability):
  - Success rate of generated test sets (percentage of tests passing Pytest validation per program and temperature).
  - Correction attempts required for a valid test set.
  - Total and average cost of generation of successful and complete generated test suite.
- For RQ2 (Adequacy and Comparison):
  - Average decision coverage percentage per program and temperature; compared to Pynguin and pre-existing test sets.
  - Average MutPy and Cosmic-Ray mutation scores per program and temperature, compared to baseline scores.
  - Standard deviation of coverage and mutation scores across temperature settings.
- For RQ3 (Fault Detection):
  - Mutation score per operator from MutPy and Cosmic-Ray (ratio of killed mutants to total mutants per operator).

[3] https://github.com/mutpy/mutpy
[4] https://github.com/sixty-north/cosmic-ray

- Identification of operators with high ($ge0.90$) and low ($le0.10$) mutation scores for LLM-generated test sets.
- Coverage and detection rates for hard-to-detect faults (e.g., comparison operator mutations, control flow mutations).

The primary aim of this experiment is to evaluate the ability of LLMs to generate test cases autonomously. We decided to use ChatGPT, specifically the 3.5-turbo model, due to its popularity, use in other studies [17], and affordable cost. Other advanced models, like ChatGPT 4.1[5] or its variants, and Claude Sonnet 4[6], may yield better performance but are considerably more expensive than 3.5-turbo. Moreover, our limited computational resources prevent using open-source models like Llama 4[7] locally.

For this purpose, we selected 40 Python projects from GitHub repositories[8][9][10]. The selection criteria were: 1) implemented in Python and 2) containing a pre-existing test set in UnitTest[11] or Pytest[12]. After standardizing the projects for the experiment, we made the dataset available in a repository.

Table 1 lists the programs used in this study, assigning a unique identification (ID) to each. The columns labeled PROGRAM, PA, CL/MO, ME/FU, CC, and SLOC represent the name of each program, the development paradigm (structural – ST or object-oriented – OO), the number of classes or modules, the number of methods or functions, maximum cyclomatic complexity, and the number of uncommented source lines of code. Additionally, these projects include pre-existing test sets created by the repository authors, which will serve as a baseline for subsequent comparative analysis.

In general, we observe that the dataset contains 22 Python programs implemented using the Structural Programming paradigm and 18 following the Object-Oriented paradigm. These programs have around 4.3 methods or functions, with an average cyclomatic complexity of 4.6 and 32.2 lines of code. It could be argued that the selected programs do not fully represent those typically found in real-world applications. However, it is essential to note that we are evaluating test generation efficiency for unit testing (at the method or function level). These programs feature various implementations of classical data structures and sorting algorithms, making them well-suited for educational purposes within an experimental framework focused on software testing and mutation analysis. These programs, with varying complexity levels, serve as a baseline for evaluating LLM performance in test generation. We assume that if test generation performs poorly on this set, it is unlikely to perform better on larger, more complex programs.

Figure 1 illustrates the flow of the experimentation. Test generation will be carried out using the OpenAI API, with prompts designed to request the creation of Python test programs in the Pytest[13] format. Each generated test will be executed to verify correctness. Should the generated tests fail to run or produce errors, feedback will be provided to the LLM so they can revise and correct the test cases. This process will repeat up to three times. If the generated test fails validation after the third attempt, it will be recorded as an error, and this outcome will be logged to contribute to the final statistical analysis of test generation success rates. The same iterative validation process applies when using Pytest to check test execution outputs. Successful test cases will exit the feedback loop if validation is confirmed.

Table 1: Set of subjects: Python programs

ID	PROGRAM	TY	CL/MO	ME/FU	CC_{MAX}	SLOC
P01	breadth_first_search.py	ST	1	2	11	39
P02	binheap.py	OO	1	5	5	38
P03	binary_search_tree.py	OO	1	7	4	54
P04	bst2.py	OO	2	21	7	152
P05	combinations.py	ST	1	2	5	26
P06	complement.py	ST	1	1	3	8
P07	convert_base.py	ST	1	2	7	14
P08	deque.py	OO	1	8	2	22
P09	depth_first_search.py	ST	1	1	11	24
P10	dijkstras.py	ST	1	1	7	27
P11	dll.py	OO	2	9	8	83
P12	edit_distance.py	ST	1	1	16	46
P13	factorial.py	ST	1	1	2	5
P14	fenwick_tree.py	OO	1	4	3	23
P15	fibonacci.py	ST	1	2	3	11
P16	find_duplicates.py	ST	1	3	5	30
P17	first_missing_positive.py	ST	1	1	6	13
P18	fizz_buzz.py	ST	1	1	5	2
P19	graph1.py	OO	4	16	4	85
P20	graph2.py	OO	1	10	3	28
P21	hamming_ops.py	ST	1	2	2	9
P22	hash_map.py	ST	1	7	4	42
P23	heap.py	OO	1	9	4	50
P24	heapsort.py	ST	1	3	4	24
P25	linked_list1.py	OO	1	5	8	57
P26	linked_list2.py	OO	2	8	6	50
P27	longest_common_prefix.py	ST	1	2	3	21
P28	lru_cache.py	OO	2	6	3	47
P29	mergesort.py	ST	1	1	1	24
P30	naive_tree.py	OO	1	6	5	49
P31	permutations.py	ST	1	1	1	10
P32	priority_queue1.py	OO	1	5	3	32
P33	priority_queue2.py	OO	1	4	2	17
P34	a_queue.py	OO	1	5	2	15
P35	quicksort.py	ST	1	1	1	17
P36	rabin_karp_substring_search.py	ST	1	1	8	18
P37	radixsort.py	ST	1	1	2	19
P38	stack.py	OO	1	3	2	8
P39	trie.py	OO	1	3	4	31
P40	word_squares.py	ST	1	1	2	18
AVG			1.2	4.3	4.6	32.2
SD			0.5	4.3	3.1	27.4

In addition, the experiment will assess different temperatures for ChatGPT (temperature refers to the randomness in the response, where higher values result in more varied outputs and lower values yield more deterministic responses). Eleven different temperature settings ranging from 0.0 to 1.0, with intervals of 0.1, will be tested to identify the optimal setting for test generation. For each temperature, 30 test sets will be generated to ensure statistical relevance, resulting in 330 test sets per Python program.

The first metric to be evaluated will be the LLM's ability to generate test cases that pass validation using Pytest. The second metric will be the decision coverage of each test set, as measured using Python's coverage tool. The third metric will be mutation score, using MutPy and Cosmic-Ray mutation tools. These tools were selected based on the study by Guerino et al. [6], which conducted a static and dynamic analysis of these and two other mutation testing tools for Python.

To draw comparisons with current automated test generation tools for Python, we will obtain both the mutation score and coverage of the tests generated by Pynguin. This will involve using a

[5]https://platform.openai.com/docs/models/gpt-4.1
[6]https://www.anthropic.com/claude/sonnet
[7]https://ollama.com/library/llama4
[8]https://github.com/clair3st/Data-Structures
[9]https://github.com/keon/algorithms
[10]https://github.com/exterkamp/Python-Data-Structures
[11]https://docs.python.org/3/library/unittest.html
[12]https://pytest.org/
[13]https://docs.pytest.org

test set that combines the ones generated by Pynguin's algorithms, which employ some artificial intelligence techniques: Dynamosa, Mio, Mosa, and Whole-Suite. Additionally, we will collect the mutation and coverage data from the original tests set that accompany the selected programs.

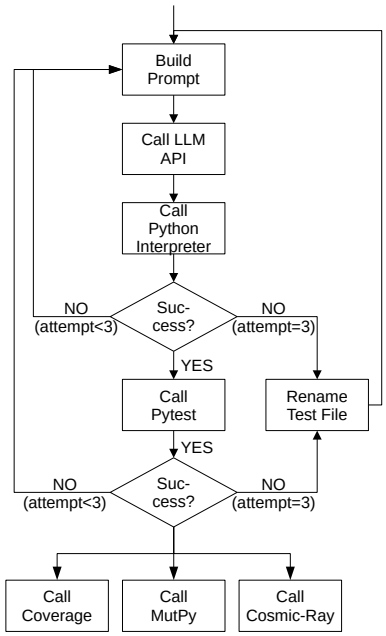

Figure 1: Flow of Experimentation

5 Data Collection and Analysis

Regarding data collection, Python scripts hosted in our GitHub repository[14] were employed to generate test cases via the OpenAI API. The system involved sending a prompt to the API and receiving a test set in response, which was then converted into a `.py` file. Listing 1 is the prompt used for test set generation. The variables used in the prompts are as follows: `{clazz}` represents the module to be tested, `{message_path_import}` includes any additional import paths required for the tests to function, and `{code}` contains the code itself.

```
{"role": "system", "content": "You are a senior software
  tester specialized in mutation testing."},
{"role": "user", "content": f"I am using mutation
  testing on the Python code below. I want to create
  tests that cover every module of the code. Do not leave
  any method untested. Give me more than one test per
  method. As a senior software tester, give me a set of
  test cases in the Pytest format. Correctly import the
  file {clazz} and its modules being tested. Utilize
  {message_path_import} to correctly import the file.
  Please, no additional information. Just the test cases.
  \n\n{code}"}
```

Listing 1: LLM Prompt for Initial Test Suite Generation

To automatically fix test cases execution errors, we used the prompt of Listings 2, where `{generated_tests}` are the initial tests generated by the LLM, and `{error_message}` is the output returned by the terminal when an error occurs.

```
{"role": "user", "content": f"The following Pytest test
  set results in these error messages. \n##########\n
  {generated_tests} \n##########\n.  You must keep all the
  correct tests and fix the assertion in the failed tests
  according to the error below. \n\nError Message:
  {error_message} \n Give me the full test set with no
  comments, only the full Python code."}
```

Listing 2: LLM Prompt for error feedback

This generated file was subsequently executed using the standard Python command. If the generated code was incorrect, feedback was sent back to the API requesting corrections, with up to three correction attempts using the prompt. If these attempts were unsuccessful, the tests were renamed and saved as `.py.err` files and logged as errors in a table. If successful, the test case was executed with Pytest to verify if the test set passed entirely; the same correction procedure was applied should this step fail.

This generation loop was repeated 30 times for each temperature setting, ranging from 0.0 to 1.0 in intervals of 0.1, yielding 330 test sets per program. The success rate of test set generation using the ChatGPT 3.5 API is provided in Table 2. Each line of the table represents a single program. The columns contain the percentage values of correctly generated tests (tests that pass the Pytest verification stage) for each temperature. The last two columns show the average success rate for that program and the average success rate excluding zeroed values, respectively. It is relevant to note that, throughout all result tables, the values of zero and 100% were highlighted in red and yellow, respectively.

Data was also collected regarding the number of tokens[15], the number of correction attempts for each test set file (whether successful or unsuccessful), and execution time, all available in the research's GitHub repository. A partial view of the token costs for the API calls is in Table 3 (complete data is available at the repository). The data follow the same logic as the previous table, where each line represents a program, and each column represents the total token cost for each temperature, considering all messages sent and received during the test generation process.

With the LLM-generated tests in place, the next step involved using various Python and Bash scripts (also available in the repository) to execute each generated test individually through the Coverage, MutPy, and Cosmic-Ray tools. This process enabled us to obtain individual coverage and mutation scores for each test set, facilitating comparison with our baseline. In the case of test sets generated by LLM, the data in the tables represent the average results obtained across all successful test sets.

As mentioned, the baseline tests are generated by Pynguin's intelligent algorithms, as well as the pre-existing tests within the programs. These baselines also underwent analysis through coverage and mutation, with their scores subsequently compared to the average scores of the LLM-generated test sets.

The coverage information, as well as mutation scores from MutPy and Cosmic-Ray, are presented in Tables 4, 5, and 6, respectively. Each row in the coverage section displays the average branch coverage values for each temperature. In the case of Pynguin and the

[14] https://github.com/aurimrv/python_experiments2/

[15] We use the OpenAI API to get the token usage estimation https://help.openai.com/en/articles/6614209-how-do-i-check-my-token-usage.

Table 2: Success rate for ChatGPT-3.5-Turbo test generation

ID	0.0	0.1	0.2	0.3	0.4	0.5	0.6	0.7	0.8	0.9	1.0	AVG Sucess Rate	AVG Sucess Rate w/o zeroes
P01	96.67	93.33	86.67	96.67	86.67	83.33	90.00	73.33	76.67	76.67	86.67	86.06	86.06
P02	0.00	0.00	0.00	0.00	3.33	3.33	6.67	6.67	3.33	10.00	0.00	3.03	5.56
P03	0.00	0.00	0.00	0.00	0.00	10.00	3.33	6.67	10.00	10.00	3.33	3.94	7.22
P04	100	100	100	100	100	96.67	96.67	90.00	80.00	73.33	93.33	93.64	93.64
P05	96.67	83.33	76.67	80.00	66.67	63.33	63.33	56.67	53.33	46.67	43.33	66.36	66.36
P06	63.33	73.33	83.33	50.00	60.00	46.67	40.00	60.00	50.00	50.00	56.67	57.58	57.58
P07	0.00	3.33	6.67	13.33	13.33	13.33	16.67	13.33	23.33	10.00	10.00	11.21	12.33
P08	6.67	3.33	13.33	13.33	10.00	10.00	6.67	13.33	13.33	13.33	16.67	10.91	10.91
P09	96.67	93.33	90.00	93.33	96.67	96.67	90.00	70.00	63.33	76.67	66.67	84.85	84.85
P10	0.00	0.00	0.00	0.00	0.00	3.33	3.33	3.33	6.67	3.33	10.00	2.73	5.00
P11	23.33	23.33	16.67	23.33	26.67	46.67	30.00	16.67	33.33	16.67	30.00	26.06	26.06
P12	0.00	0.00	0.00	0.00	0.00	0.00	0.00	0.00	0.00	0.00	0.00	0.00	0.00
P13	100	100	100	96.67	100	100	100	100	93.33	93.33	90.00	97.58	97.58
P14	0.00	3.33	3.33	6.67	3.33	6.67	3.33	3.33	0.00	0.00	6.67	3.33	4.58
P15	100	100	100	100	100	100	100	93.33	96.67	93.33	93.33	97.88	97.88
P16	100	100	100	100	96.67	90.00	76.67	56.67	36.67	56.67	63.33	79.70	79.70
P17	100	100	100	100	100	100	96.67	100	100	96.67	100	99.39	99.39
P18	100	100	100	100	96.67	100	100	96.67	100	100	90.00	96.03	96.03
P19	36.67	36.67	16.67	16.67	20.00	26.67	23.33	26.67	36.67	20.00	20.00	25.45	25.45
P20	0.00	0.00	3.33	3.33	6.67	13.33	13.33	3.33	16.67	13.33	10.00	7.58	9.26
P21	60.00	83.33	63.33	66.67	73.33	70.00	73.33	76.67	76.67	63.33	66.67	70.30	70.30
P22	30.00	20.00	23.33	20.00	30.00	43.33	56.67	36.67	56.67	63.33	50.00	39.09	39.09
P23	36.67	33.33	23.33	30.00	33.33	26.67	30.00	10.00	23.33	6.67	30.00	25.76	25.76
P24	0.00	0.00	0.00	0.00	3.33	6.67	33.33	33.33	40.00	26.67	43.33	16.97	26.67
P25	100	93.33	70.00	60.00	36.67	43.33	53.33	43.33	43.33	40.00	50.00	57.58	57.58
P26	6.67	23.33	10.00	13.33	26.67	23.33	43.33	53.33	50.00	33.33	40.00	29.39	29.39
P27	96.67	100	100	100	96.67	100	96.67	100	100	93.33	96.67	98.18	98.18
P28	93.33	80.00	73.33	60.00	63.33	46.67	50.00	53.33	36.67	60.00	63.33	61.82	61.82
P29	43.33	23.33	46.67	43.33	56.67	56.67	56.67	56.67	56.67	46.67	46.67	48.48	48.48
P30	0.00	0.00	0.00	0.00	0.00	0.00	10.00	0.00	6.67	3.33	10.00	2.73	7.50
P31	3.33	3.33	20.00	16.67	26.67	30.00	26.67	26.67	40.00	16.67	26.67	21.52	21.52
P32	0.00	0.00	0.00	3.33	3.33	10.00	6.67	3.33	13.33	13.33	6.67	5.45	7.50
P33	96.67	93.33	86.67	80.00	76.67	56.67	76.67	63.33	83.33	56.67	56.67	75.15	75.15
P34	100	100	100	93.33	86.67	86.67	90.00	90.00	80.00	86.67	83.33	90.61	90.61
P35	96.67	100	96.67	96.67	86.67	100	86.67	90.00	73.33	83.33	66.67	88.79	88.79
P36	6.67	6.67	20.00	6.67	13.33	40.00	26.67	10.00	23.33	33.33	46.67	21.21	21.21
P37	0.00	0.00	0.00	0.00	0.00	0.00	3.33	0.00	0.00	3.33	0.00	0.61	3.33
P38	0.00	0.00	3.33	10.00	0.00	0.00	6.67	6.67	3.33	6.67	3.33	3.64	5.33
P39	0.00	0.00	0.00	0.00	0.00	3.33	6.67	6.67	3.33	13.33	20.00	4.85	8.89
P40	0.00	0.00	0.00	0.00	0.00	0.00	3.33	0.00	0.00	3.33	0.00	0.61	3.33
AVG	33.34	23.33	21.67	21.67	28.34	41.67	36.67	35.00	38.34	33.33	43.33	27.73	43.97
SD	44.85	44.28	42.2	41.06	39.43	37.77	36.25	35.87	33.35	33.41	32.83	37.26	35.85

Table 3: Total token costs for ChatGPT-3.5-Turbo test generation

ID	0,0	0,1	0,2	0,3	0,4	0,5	0,6	0,7	0,8	0,9	1,0	SUM
P01	37,654	41,493	48,441	37,280	48,871	52,200	44,164	61,390	56,862	57,092	47,296	532,743
P02	117,984	119,915	120,832	120,464	118,813	130,534	124,920	125,676	119,327	116,866	126,034	1,341,365
P03	122,036	125,327	124,792	121,635	123,597	114,534	121,006	109,026	112,058	114,723	120,862	1,309,596
P04	76,410	76,268	76,156	80,126	76,287	85,698	91,644	97,793	121,442	127,973	107,003	1,016,800
P05	31,964	43,884	49,308	49,319	55,324	61,589	66,262	67,033	75,940	66,382	78,804	645,809
...						...						...
P12	154,017	215,690	163,955	146,097	185,866	147,638	137,502	158,579	131,509	151,758	135,194	1,727,805
P13	12,227	12,207	12,333	14,478	12,482	12,491	12,627	13,378	16,671	15,978	19,641	154,513
...						...						...
P28	66,951	84,859	90,918	120,261	106,519	129,173	128,404	119,671	141,580	111,556	114,048	1,213,940
P29	110,408	132,269	106,433	100,042	88,492	87,168	91,178	90,632	89,440	93,424	97,458	1,086,944
P30	142,685	148,663	150,004	145,668	145,456	148,806	130,862	130,355	135,922	146,731	133,870	1,559,022
P31	420,332	433,404	218,054	198,090	214,811	205,959	106,458	116,770	99,189	103,064	103,074	2,219,205
P32	144,507	134,558	128,340	118,286	119,909	118,067	118,839	130,309	117,661	106,077	119,895	1,356,448
P33	23,179	26,142	29,324	36,817	37,872	58,351	38,226	54,181	34,705	58,080	62,382	459,259
P34	20,622	20,615	20,692	26,107	31,548	32,166	30,363	28,634	36,126	32,907	33,536	313,316
P35	31,756	27,497	31,224	30,671	43,435	26,699	43,882	38,084	59,740	46,377	65,843	445,208
P36	104,324	96,941	90,173	100,329	91,226	78,821	83,048	90,724	90,948	86,741	74,179	987,454
P37	128,793	130,451	129,370	130,996	135,692	133,256	130,332	124,633	127,874	123,464	128,704	1,423,565
P38	112,440	109,416	106,142	80,128	94,434	95,744	87,254	94,865	92,124	90,180	92,428	1,055,155
P39	119,219	114,696	116,654	116,766	118,873	117,267	112,429	118,365	116,652	108,801	105,592	1,265,314
P40	133,871	138,090	136,055	128,402	129,288	118,699	118,554	115,005	117,461	109,288	114,763	1,359,476
SUM	3,813,172	3,894,842	3,710,459	3,680,756	3,723,089	3,640,839	3,473,508	3,644,665	3,595,037	3,686,263	3,641,439	**40,504,069**
AVG	95,329.3	97,371.1	92,761.5	92,018.9	93,077.2	91,021.0	86,837.7	91,116.6	89,875.9	92,156.6	91,036.0	1,012,601.7
SD	74,096.3	76,492.2	54,389.9	52,451.8	52,183.9	48,841.2	42,508.5	43,706.4	40,174.4	42,759.8	39,348.2	530,024.8

pre-existing tests, only the absolute coverage value is provided, as only a single test set is considered. For the mutation tools, the rows show the average mutation scores per tool, following the same logic of using averages for the LLM-generated tests and absolute values for Pynguin and the pre-existing test sets.

It is important to note that some of the averages for the test sets generated by the LLM may lack precision due to cases where very few tests were successful—those with low percentages in Table 2.

Notably, there were a few exceptional cases in this test generation process. First, ChatGPT was unable to generate valid tests for the program `edit_distance.py` (P12), which was the only instance where this occurred, despite cases where other programs had fewer generated test sets. Another exception involved the program `stack.py` (P38), for which Cosmic-Ray could not create mutants. Upon analyzing the program, this likely occurred because this stack implementation is an extension of the `linked_list2.py` program, meaning there are no lines of code that Cosmic-Ray deemed suitable for generating mutants.

6 Discussion

Test sets generation success and cost

Regarding the success rate of test generation by ChatGPT-3.5-turbo, we observed considerable variability in the results, as shown in Table 2. For some programs, we achieved multiple instances of 100% efficiency in creating tests that pass Pytest across various temperature settings. However, when analyzing average generation values, they generally remain below 28% of success. This is primarily due to the presence of numerous programs for which the LLM failed to generate passing tests, resulting in multiple cases with 0% success or very low success percentages.

Considering the different temperature settings for test generation, we find that less deterministic responses (temperatures of 0.5 and above) yield more successful tests on average. Notably, temperature settings of 0.5 and 1.0 stand out, exceeding the average by 41.67% and 43.33%, respectively.

OpenAI API charges are calculated per million tokens, and prices vary by model. For ChatGPT-3.5-turbo, the cost is $ 0.50 per million of input tokens and $ 1.50 per million of output tokens, which results in a cost per token of $ 0.0000005 per input token and $ 0.0000015 per output token. We will use the average cost per token of $ 0.0000010 in our cost estimations.

Thus, considering the total cost of our experiment, 40,504,069 tokens were used to generate all tests, resulting in an approximate cost of $ 40.50. To estimate the price of a successful test set in our experiment, we can consider one of the programs with a 100% success rate, such as `factorial.py` (P13) at temperature 0.5. By dividing the total tokens used for that ($12,491/30$), we obtain approximately 416 tokens per successful test set, corresponding to a cost of roughly $ 0.000416.

In a worst-case scenario, we need to generate test sets 330 x 6 times (due to failures in Python test generation and test set rejections by Pytest), and we would incur a significantly higher cost. Although this did not occur in our experiment, we can use the `edit_distances.py` program (P12) as an example, as it did not yield any successfully generated tests. For instance, at a temperature of 0.5, the cost per test was $ 0.004921, accounting for all correction attempts.

> **Answering RQ1**, the ChatGPT-3.5-turbo model allows us to create valid test sets for 39 out of 40 programs for at least one temperature parameter at a reasonable cost, especially if we consider the generation of a single test set for a given temperature. The average success rate is below 28%, which means that, in general, this model created two success test cases for every ten it generated. Hopefully, the cost is significantly low. Even in the worst scenario, considering the generation of the test set incrementally, using all temperatures several times (330 test sets in our experiment), the total cost was around $ 40.50.

Test sets adequacy and comparison

Regarding the decision coverage (Table 4), the higher the test set's successful rates, the better the decision coverage. The high standard deviation highlights significant variation in the average scores across different temperatures among various programs. The temperature with the best results in terms of decision coverage is 0.6, with an average coverage of 86.10% and a lower standard deviation of 20.88.

Now that we have the baseline tests for comparison, we can infer that within the scope of our experiment, the test set generated by ChatGPT-3.5-turbo for each individual temperature does not achieve the performance of the automatic test generation tool for Python, nor that of the custom-designed tests for each program. The decision coverage from Pynguin and the pre-existing tests stand out, being 4.65% and 8.74% above the best average coverage score (86.10%), respectively, while being 34.39% and 38.48% above the worst-performing average (56.36%).

However, as can be observed in the column "LLM ALL," which merges all generated test sets for all temperatures, the coverage improves to a superior level compared to Pynguin and Pre-existing test sets for almost all subject programs, with a few exceptions. This suggests that varying temperatures in the LLM model is crucial for generating diverse test cases, enhancing test set coverage, and improving fault detection capability. The final average coverage for this test set is 95.9, 1.1% above the average coverage of Pre-existing test cases.

Regarding the mutation score, we must observe that we did not determine the possible equivalent mutants among the live ones. In this sense, mutation scores can be higher for all test sets after equivalent mutant determination. However, once we use the same set of mutants, our comparison is fair, as long as possible equivalent ones remain alive for all test sets.

We begin by analyzing MutPy (Table 5), where we note a sharp drop in 100% scores, though averages become more comparable between LLM-generated and baseline tests. Pre-existing tests exceed the best LLM average by only 2.03%, while Pynguin falls between LLM temperatures. This indicates that, despite challenges in generating valid tests and achieving full decision coverage, considering the test set of each temperature individually, LLM-generated sets can still be effective in killing MutPy mutants.

Table 4: Decision coverage

ID	LLM - Temperatures											LLM	Non-LLM	
	0.0	0.1	0.2	0.3	0.4	0.5	0.6	0.7	0.8	0.9	1.0	ALL	Pynguin	Pre-existing
P01	90.3	89.9	90.2	89.5	89.3	90.4	89.1	87.8	87.2	88.7	86.7	100	63.6	63.6
P02	0.0	0.0	0.0	0.0	94.4	94.4	83.3	66.7	72.2	79.6	0.0	96.0	94.4	94.4
P03	0.0	0.0	0.0	0.0	0.0	90.0	90.0	82.5	88.3	85.0	90.0	97.0	100	100
P04	74.0	73.0	72.6	72.3	73.0	74.0	71.1	72.5	74.3	70.6	64.9	79.0	88.8	88.8
P05	100	100	100	100	100	100	100	100	100	100	100	100	100	100
P06	100	100	100	100	100	100	100	100	100	100	100	100	100	100
P07	0.0	100	100	93.8	84.4	100	95.0	71.9	87.5	75.0	87.5	100	100	100
P08	100	100	100	100	100	83.3	100	100	100	81.3	90.0	100	100	100
P09	100	99.5	99.7	99.2	99.5	99.5	99.5	100	98.5	97.8	96.4	100	100	100
P10	0.0	0.0	0.0	0.0	0.0	91.7	91.7	91.7	87.5	91.7	83.3	97.0	0.0	92.9
P11	62.7	63.9	60.6	64.7	60.8	66.7	63.9	68.9	61.9	65.6	67.6	99.0	97.2	97.2
P12	0.0	0.0	0.0	0.0	0.0	0.0	0.0	0.0	0.0	0.0	0.0	0.0	86.4	88.5
P13	100	100	100	100	100	100	100	100	100	100	100	100	100	100
P14	0.0	100	100	100	100	100	100	100	0.0	0.0	100	100	100	100
P15	100	100	100	100	100	100	100	100	100	100	100	100	83.3	100
P16	87.5	87.5	87.5	87.5	87.5	87.7	88.0	87.9	88.6	89.3	90.8	100	100	100
P17	100	100	100	100	100	100	100	100	99.4	100	99.4	100	100	100
P18	100	100	100	100	100	100	100	100	100	100	100	100	100	100
P19	67.8	63.2	67.7	66.8	70.1	74.2	64.3	71.0	65.5	68.6	70.1	95.0	92.1	97.7
P20	0.0	0.0	25.0	25.0	25.0	25.0	37.5	25.0	47.5	46.9	25.0	100	100	100
P21	75.0	75.0	76.3	76.3	80.7	83.3	79.6	85.9	90.2	92.1	88.8	100	100	100
P22	81.3	81.8	78.6	80.3	81.8	80.1	80.0	78.1	76.5	79.4	79.4	98.0	88.9	95.5
P23	72.7	73.6	76.0	76.8	79.5	78.4	79.8	77.3	77.9	86.4	80.3	100	100	100
P24	0.0	0.0	0.0	0.0	100	100	100	100	100	100	100	100	100	100
P25	73.1	73.1	72.5	72.4	73.1	72.8	71.2	75.7	73.1	74.7	74.9	98.0	80.8	92.3
P26	72.2	75.4	77.8	76.4	73.6	77.0	74.8	78.5	75.6	77.2	75.9	100	100	100
P27	91.8	91.3	93.8	92.5	94.4	94.2	95.3	95.0	96.3	96.9	97.0	100	100	100
P28	64.3	67.2	72.2	74.3	75.0	81.3	77.5	81.3	83.0	79.9	84.2	100	100	100
P29	100	100	100	100	100	100	100	100	100	100	100	100	100	100
P30	0.0	0.0	0.0	0.0	0.0	0.0	100	0.0	100	100	100	100	21.4	100
P31	100	100	100	100	100	100	100	100	100	100	100	100	100	100
P32	0.0	0.0	0.0	70.0	70.0	70.0	70.0	70.0	70.0	70.0	70.0	88.0	90.0	90.0
P33	50.0	50.0	50.0	50.0	50.0	50.0	50.0	50.0	50.0	50.0	50.0	89.0	50.0	50.0
P34	100	100	100	100	100	100	100	100	100	100	100	100	100	100
P35	100	100	100	100	100	100	100	100	100	100	100	100	100	100
P36	91.7	91.7	93.1	91.7	93.8	93.1	92.7	94.5	91.7	92.5	92.3	100	100	100
P37	0.0	0.0	0.0	0.0	0.0	0.0	100	0.0	0.0	100	0.0	100	100	100
P38	0.0	0.0	100	100	0.0	0.0	100	100	100	100	100	100	100	100
P39	0.0	0.0	0.0	0.0	0.0	100	100	96.4	100	100	97.6	100	92.9	42.9
P40	0.0	0.0	0.0	0.0	0.0	0.0	100	0.0	0.0	100	0.0	100	100	100
AVG	56.4	61.4	64.8	66.5	68.9	76.4	86.1	77.7	78.6	83.5	78.6	95.9	90.8	94.8
SD	43.7	42.6	41.1	39.6	38.3	33.1	20.9	30.8	30.1	24.1	30.9	16.1	21.4	13.0

A similar trend occurred for the mutants generated by Cosmic-Ray (Table 6); however, overall mutation scores tend to be lower. The difference may be due to the nature of the mutants created by this tool, as was noted by the study by Guerino and Vincenzi [7], where the results for Cosmic-Ray were also lower than MutPy, which could be explored in future analyses to assess the LLM's capability to kill specific mutants or identify equivalent mutants.

The same observation from the decision coverage can be made regarding mutation score: the temperature of 0.6 produces the best average results in MutPy and Cosmic-Ray, with the lowest standard deviation. By combining all LLM-generated test sets, we also observed an improvement of around 10% considering the best mutation score from temperature 0.6, both on MutPy and Cosmic-Ray, and on average, also overcoming Pynguin and Pre-existing test sets.

Answering RQ2, the LLM performs significantly better than Pynguin in fault detection within the MutPy model and moderately better within the Cosmic-Ray model, considering test sets from individual temperature. These results suggest that Pynguin exhibits behavior similar to Java automatic test generators as observed by other researchers, particularly regarding its limited ability to kill mutants [2, 15]. They are better at determining test sets that achieve high code and decision coverage, but fail to select effective tests for detecting faults modeled by mutation operators. The combination of test cases from different temperatures is also important to improve fault detection, considering the fault models from MutPy and Cosmic-Ray.

Fault Detection Capability

Finally, we evaluated the fault detection capability of the complete set of LLM-based generated tests. Tables 7 and 8 illustrate collected data for MutPy and Cosmic-Ray, respectively. Due to space limitations, the complete data of Cosmic-Ray is available at the repository[16].

Considering the fault models of MutPy and Cosmic-Ray, represented by their mutation operators, we aim to identify mutation

[16]https://github.com/aurimrv/python_experiments2

Table 5: MutPy mutation score

ID	LLM - Temperatures											LLM	Non-LLM	
	0.0	0.1	0.2	0.3	0.4	0.5	0.6	0.7	0.8	0.9	1.0	ALL	Pynguin	Pre-existing
P01	72.5	72.5	73.0	73.6	71.1	70.2	72.4	73.1	70.5	65.9	69.4	92.1	34.9	67.4
P02	0.0	0.0	0.0	0.0	94.9	94.9	90.7	85.6	85.6	88.3	0.0	97.2	96.9	95.0
P03	0.0	0.0	0.0	0.0	0.0	88.7	88.7	85.2	86.9	84.5	88.7	88.7	77.5	91.6
P04	80.4	79.2	78.7	78.4	79.3	80.3	77.4	78.6	80.9	77.6	71.6	91.8	81.6	93.6
P05	99.8	98.9	98.8	98.2	97.7	98.1	97.6	95.8	97.6	95.2	96.7	100	32.1	45.6
P06	92.3	93.7	93.5	92.8	95.3	95.6	96.2	97.4	98.5	96.9	97.7	100	100	93.8
P07	0.0	92.9	92.9	88.1	81.6	91.7	89.1	71.4	82.7	74.6	80.2	97.5	97.6	90.9
P08	92.1	92.1	92.1	92.1	92.1	92.1	92.1	92.1	69.7	91.5	92.1	93.1	89.5	94.7
P09	80.2	79.9	80.5	79.1	77.9	79.9	79.2	80.1	80.0	77.0	77.2	95.6	71.1	82.9
P10	0.0	0.0	0.0	0.0	0.0	92.5	92.5	92.5	90.0	95.0	86.7	95.0	0.0	89.4
P11	77.1	79.4	76.3	77.9	74.4	78.4	76.7	79.4	72.7	74.8	77.1	96.7	87.7	90.9
P12	0.0	0.0	0.0	0.0	0.0	0.0	0.0	0.0	0.0	0.0	0.0	0.0	51.9	83.7
P13	100	100	100	100	100	100	100	100	100	100	100	100	100	100
P14	0.0	98.4	96.8	98.4	98.4	98.4	98.4	98.4	0.0	0.0	98.4	98.0	96.8	95.9
P15	94.7	93.5	94.0	94.2	94.5	93.8	94.3	93.9	93.6	93.6	94.1	95.0	75.0	96.7
P16	86.4	86.4	86.4	86.4	86.4	86.3	86.8	86.9	86.6	86.9	87.8	97.7	43.2	86.3
P17	92.9	92.9	92.8	91.9	92.4	92.3	91.2	91.5	91.5	91.7	89.7	96.0	88.0	90.6
P18	100	100	100	100	100	100	100	100	100	100	100	100	12.5	100
P19	76.1	74.3	72.7	72.2	71.0	78.5	73.5	74.5	71.4	75.2	78.9	93.1	84.2	92.9
P20	0.0	0.0	63.2	70.2	70.2	68.4	69.7	63.2	76.5	75.0	62.0	90.2	63.2	96.5
P21	87.5	87.5	88.2	88.1	90.3	91.7	89.8	92.9	94.8	95.7	94.1	100	100	100
P22	89.2	89.5	88.4	89.5	87.6	89.3	84.8	80.1	76.3	83.2	82.6	92.5	92.1	93.1
P23	75.7	77.3	80.8	80.9	83.0	79.7	82.4	81.8	83.3	84.9	81.5	94.0	92.1	93.9
P24	0.0	0.0	0.0	0.0	80.0	80.0	80.0	79.8	79.6	79.7	79.6	80.0	52.5	80.0
P25	77.5	77.5	77.5	76.7	77.4	77.5	78.0	78.8	79.3	80.9	80.7	97.3	90.0	95.0
P26	85.4	87.2	90.3	88.0	86.5	87.7	88.1	89.4	87.6	89.0	87.7	100	91.7	98.2
P27	93.5	93.5	93.5	93.8	94.2	93.6	93.0	93.3	92.9	93.2	91.8	96.2	92.9	90.6
P28	84.1	84.9	86.1	85.9	86.4	87.8	86.3	86.9	87.2	85.9	87.1	94.7	86.7	84.5
P29	97.2	97.2	97.2	97.2	97.2	97.2	97.2	97.2	97.2	97.2	97.2	96.9	69.4	95.1
P30	0.0	0.0	0.0	0.0	0.0	0.0	84.4	0.0	83.9	83.9	83.9	83.9	43.6	97.1
P31	93.8	93.8	93.8	92.5	93.8	93.8	91.4	90.6	92.7	92.5	93.0	92.9	37.5	94.7
P32	0.0	0.0	0.0	86.8	86.8	87.4	86.8	86.8	85.4	87.3	86.8	89.4	77.4	77.8
P33	87.4	87.5	86.7	85.9	85.9	85.8	85.3	85.5	86.0	85.5	86.8	82.4	83.3	80.7
P34	87.1	87.0	87.5	87.7	88.1	88.8	88.1	87.3	87.5	88.1	87.7	88.2	69.6	91.3
P35	91.3	91.3	91.3	91.3	91.3	91.3	91.3	91.3	91.1	91.1	91.1	91.3	52.2	50.0
P36	94.6	94.6	95.2	94.6	95.5	95.1	95.1	95.2	94.6	94.8	94.8	98.2	60.7	92.1
P37	0.0	0.0	0.0	0.0	0.0	0.0	93.6	0.0	0.0	93.6	0.0	92.9	61.3	91.4
P38	0.0	0.0	91.7	91.7	0.0	0.0	91.7	91.7	91.7	91.7	91.7	87.5	16.7	100
P39	0.0	0.0	0.0	0.0	0.0	98.2	98.2	91.1	96.4	92.9	95.2	98.0	82.1	40.0
P40	0.0	0.0	0.0	0.0	0.0	0.0	82.6	0.0	0.0	87.0	0.0	85.0	56.5	92.0
AVG	57.2	62.1	66.0	68.1	70.0	77.6	85.9	78.5	78.1	83.0	78.5	91.5	69.8	87.9
SD	43.0	41.7	39.4	37.9	36.3	30.7	15.9	27.7	27.6	20.8	27.8	15.7	26.3	14.1

operators that generate mutants easier to be killed by LLM test cases, as well as those harder to be killed by LLM. The mutation score represents the overall effectiveness of the test suite in detecting faults. In general terms, as can be observed in Tables 7 and 8, LLM test set determines higher mutation scores on MutPy than on Cosmic-Ray.

It suggests that MutPy generates mutants that are possibly inherently easier to detect and that the LLM-generated test cases are better aligned with the types of mutations that MutPy introduces. On the other hand, Cosmic-Ray introduces more subtle mutations that are harder for automated tests to detect.

Given time constraints, mutation operators that exhibit high mutation scores ($\geq$0.90) may be avoided, as they are easier to detect by the LLM-generated test set. Considering MutPy, these operators are: SIR (Slice Index Remove), LOR (Logical Operator Replacement), SDI (Statement Deletion), SVD (Self Variable Deletion), and ZIL (Zero Iteration Loop). Considering Cosmic-Ray, these operators are: Many binary operator replacements (BitAnd, Pow), ReplaceBinaryOperator_Add_Div, ExceptionReplacer, and ZeroIterationForLoop.

Based on the nature of these mutation operators, we can suggest that LLM-generated tests are particularly good at detecting:

- Loop boundary issues (ZIL/ZeroIterationForLoop);
- Exception handling problems (EHD/ExceptionReplacer);
- Statement/variable deletions (SDI/SVD);
- Certain arithmetic operator replacements, especially division-related replacements.

On the other hand, LLM-generated test sets had difficulty detecting other types of faults. Especially on the MutPy, faults modeled by EXS (Exception Swallowing), LCR (Logical Connector Replacement), and BCR (Break/Continue Replacement) reached very low mutation scores. The same for Cosmic-Ray, considering the mutation operators ReplaceComparisonOperator_Is_Eq, ReplaceComparisonOperator_Eq_Is, ReplaceComparisonOperator_IsNot_NotEq, ReplaceComparisonOperator_NotEq_IsNot, ReplaceAndWithOr, ReplaceBreakWithContinue, many of which have scores of 0.0, indicating that they require very specific test cases to be detected.

An in-depth data analysis reveals interesting patterns:

Table 6: Cosmic-Ray mutation score

ID	LLM - Temperatures											LLM	Non-LLM	
	0.0	0.1	0.2	0.3	0.4	0.5	0.6	0.7	0.8	0.9	1.0	ALL	Pynguin	Pre-existing
P01	53.8	54.8	55.1	56.1	53.7	54.5	56.5	56.7	53.3	49.1	53.9	85.0	41.1	82.8
P02	0.0	0.0	0.0	0.0	94.2	95.3	92.1	88.3	89.5	91.4	0.0	97.7	98.3	97.1
P03	0.0	0.0	0.0	0.0	0.0	87.3	91.2	83.8	86.3	79.4	85.3	91.2	88.2	91.2
P04	69.6	69.2	69.2	69.1	69.2	69.7	68.6	69.0	70.4	68.3	63.7	86.8	77.5	86.0
P05	98.8	98.4	98.3	98.4	98.5	98.4	98.4	98.4	98.4	98.2	98.4	100	48.3	100
P06	96.3	96.3	96.7	96.3	97.2	100	100	98.6	100	100	99.1	100	100	96.3
P07	0.0	90.4	92.0	87.0	81.8	88.6	87.8	74.6	83.0	76.8	80.3	96.0	94.4	91.2
P08	100	100	100	100	100	100	100	100	75.0	100	100	100	100	100
P09	62.0	62.5	62.5	60.2	59.5	61.0	60.4	61.2	62.5	60.9	61.9	94.0	60.8	85.5
P10	0.0	0.0	0.0	0.0	0.0	82.9	80.0	82.9	78.6	85.7	79.1	91.4	0.0	88.6
P11	55.0	55.3	53.2	57.3	52.9	56.4	53.3	54.8	51.2	54.2	53.2	91.9	87.4	83.8
P12	0.0	0.0	0.0	0.0	0.0	0.0	0.0	0.0	0.0	0.0	0.0	0.0	60.8	85.6
P13	94.6	94.4	95.7	97.3	97.0	97.4	97.4	98.2	96.6	96.6	98.6	100	100	97.2
P14	0.0	99.3	84.5	99.3	99.3	98.6	97.9	98.6	0.0	0.0	97.2	99.3	96.5	96.5
P15	99.8	99.2	99.4	99.3	99.2	98.9	98.7	98.5	98.6	98.4	98.6	100	56.8	100
P16	69.4	69.4	69.4	69.4	69.5	69.6	70.6	70.7	71.3	72.2	71.8	87.1	43.6	71.0
P17	88.0	88.0	88.9	90.5	85.5	91.7	84.5	87.2	89.9	89.2	86.8	96.4	83.3	88.1
P18	97.1	97.1	97.1	97.1	97.1	97.1	97.1	97.0	96.9	97.1	97.1	97.1	17.4	94.2
P19	65.2	59.4	56.5	56.1	55.1	67.0	57.1	60.8	52.9	63.9	66.8	89.8	89.8	89.8
P20	0.0	0.0	33.3	33.3	33.3	33.3	33.3	33.3	53.3	41.7	33.3	100	66.7	100
P21	84.6	85.5	84.6	86.7	89.1	94.9	88.1	92.3	94.0	93.9	89.1	100	100	96.6
P22	91.1	91.3	90.3	91.3	86.9	90.8	83.1	73.5	67.9	78.5	82.6	94.8	97.1	93.6
P23	62.6	62.6	66.4	74.8	68.9	57.0	80.7	0.0	82.2	70.7	71.6	91.6	87.9	86.6
P24	0.0	0.0	0.0	0.0	85.7	85.7	85.7	85.7	85.7	85.9	85.9	85.7	66.7	85.7
P25	75.0	75.0	75.1	73.9	74.6	73.6	72.4	76.1	76.2	79.6	80.0	93.2	77.3	93.2
P26	85.4	85.1	85.7	85.4	84.9	85.1	83.9	87.4	85.7	86.1	83.9	95.8	84.7	95.8
P27	84.6	85.5	85.0	85.6	86.1	84.9	83.3	82.5	83.0	82.3	81.3	93.3	83.3	86.7
P28	70.9	73.1	75.0	74.8	75.9	80.5	78.0	77.2	79.7	77.8	79.2	100	100	73.1
P29	98.0	98.0	98.0	98.0	98.0	98.0	98.0	98.0	98.0	98.0	98.0	98.0	56.0	96.0
P30	0.0	0.0	0.0	0.0	0.0	0.0	24.1	0.0	24.1	24.1	24.1	24.1	86.2	100
P31	100	100	100	100	100	100	100	87.5	100	100	100	100	87.5	100
P32	0.0	0.0	0.0	47.5	47.5	46.7	47.5	47.5	46.3	47.5	47.5	50.0	40.0	47.5
P33	84.6	84.6	84.6	84.6	84.6	84.6	84.6	84.2	84.3	84.6	85.5	92.3	61.5	84.6
P34	100	100	100	100	96.2	100	96.3	92.6	83.3	88.5	92.0	100	100	100
P35	88.3	88.3	88.2	88.0	88.0	88.1	88.0	87.8	87.6	87.7	87.7	88.3	60.0	88.3
P36	85.0	86.5	86.5	85.9	87.3	87.0	86.4	87.2	86.4	86.3	86.0	92.9	53.0	86.1
P37	0.0	0.0	0.0	0.0	0.0	0.0	92.1	0.0	0.0	92.1	0.0	92.1	60.3	85.7
P38	0.0	0.0	0.0	0.0	0.0	0.0	0.0	0.0	0.0	0.0	0.0	0.0	0.0	0.0
P39	0.0	0.0	0.0	0.0	0.0	93.1	93.1	91.4	89.7	92.2	93.1	93.1	41.4	10.3
P40	0.0	0.0	0.0	0.0	0.0	0.0	87.5	0.0	0.0	87.5	0.0	87.5	79.2	87.5
AVG	54.0	58.7	59.3	61.1	64.9	72.4	76.9	69.1	69.0	74.2	69.8	86.9	70.8	85.6
SD	41.9	40.9	39.8	39.0	36.6	32.2	25.6	33.1	31.3	27.7	32.1	24.4	26.9	21.1

(1) Asymmetric operator replacements: Replacing operator A with B often has a different detection rate than replacing B with A. For example:
   - Replacing == with `is` has a 0.02 score
   - Replacing `is` with == has a 0.00 score
(2) Comparison operator sensitivity: LLM tests seem particularly weak at detecting subtle comparison changes, especially those involving identity checks;
(3) Arithmetic vs. logical operators: Tests are generally better at catching arithmetic operator mutations (AOR: 0.83) than logical connector mutations (LCR: 0.53);
(4) Control flow sensitivity: Mutations that affect program flow subtly (like replacing `break` with `continue` or vice versa) are harder to detect than those that cause more obvious failures.

A final observation is that among live mutants, some may be equivalent. Future work will involve manually identifying them to improve analysis precision.

Answering RQ3, the LLM-generated tests may focus on the "happy path" and common error cases while missing subtle semantic differences that don't immediately cause obvious failures. The tests appear to be less effective at detecting mutations that change program behavior in ways that require a deep understanding of Python's semantics, particularly in relation to object identity, control flow, and exception handling. These results suggest that for more robust testing, LLM-generated tests should be complemented with manually crafted tests that specifically target the weak areas identified. Alternatively, we can develop an incremental strategy and create specific prompts, considering these types of potential faults when generating tests.

7 Threats to Validity

Several threats to validity should be considered when evaluating the capability of LLMs to generate tests autonomously.

As previously mentioned, the 40 code structures selected for analysis represent only a tiny subset of the numerous configurations commonly found in real-world applications.

Table 7: Fault Model from MutPy Mutation Operators

Operator	Dead	Alive	Score
AOD	815	118	0.87
AOR	17255	3412	0.83
ASR	4490	933	0.83
BCR	638	375	0.63
CDI	13511	2765	0.83
COD	6545	1040	0.86
COI	27731	4162	0.87
EHD	748	68	0.92
EXS	51	765	0.06
LCR	2308	2010	0.53
LOR	1174	79	0.94
OIL	9896	2450	0.80
RIL	5114	2035	0.72
ROR	20798	7885	0.73
SDI	15109	1167	0.93
SDL	72931	16978	0.81
SIR	1321	19	0.99
SVD	39052	4261	0.90
ZIL	11116	1220	0.90

Table 8: Fault Model from Cosmic-Ray Mutation Operators

Operator	Dead	Alive	Score
AddNot	25146	4013	0.86
ExceptionReplacer	748	68	0.92
NumberReplacer	51222	28116	0.65
ReplaceAndWithOr	1029	1651	0.38
ReplaceBinaryOperator_Add_Div	5861	186	0.97
ReplaceBinaryOperator_BitAnd_Add	22	0	1.00
...	...	...	...
ReplaceBinaryOperator_Pow_Sub	41	0	1.00
ReplaceBinaryOperator_Sub_Div	4510	83	0.98
ReplaceBreakWithContinue	62	103	0.38
ReplaceComparisonOperator_Eq_Is	81	4483	0.02
ReplaceComparisonOperator_Is_Eq	1	354	0.00
ReplaceComparisonOperator_IsNot_NotEq	0	760	0.00
ReplaceComparisonOperator_NotEq_IsNot	0	394	0.00
...	...	...	...
ReplaceOrWithAnd	2213	389	0.85
ReplaceTrueWithFalse	345	183	0.65
ZeroIterationForLoop	6687	462	0.94

This sample may not capture the full diversity and complexity of Python applications, limiting the generalizability of our findings. More advanced generative models than ChatGPT-3.5-turbo could improve automated test generation, but cost and scope guided the choice of this model, as this study serves as an initial exploration of fully automated generative testing for Python.

Additionally, the pre-existing tests used as a baseline are not guaranteed to have been created manually or using test generation tools; they serve only as a baseline because they were available in the same dataset from which the programs were extracted.

Finally, further refinement and experimentation with varied prompts could improve results, as this study used a single prompt for generation and another for correction. Additionally, using only one LLM model is also a limitation, since multiple models could reveal performance variability. Still, ChatGPT-3.5-turbo performed impressively for its cost, especially when combining test cases from different temperatures to enhance diversity.

8 Conclusion

The findings of this study highlight the mixed effectiveness of ChatGPT-3.5-turbo in generating valid Python test cases. For simpler code, it consistently produced passing tests, sometimes reaching 100% at certain temperatures However, when averaged across all programs, the success rate of generated tests fell below 28%, mainly due to numerous cases in which the model failed to produce passing tests. This inconsistency indicates that while LLMs show promise for autonomous test generation, they struggle with more complex or less deterministic applications.

The impact of temperature on test generation success was notable. Less deterministic settings (0.5 and above) generally yielded more functional tests, with the highest success rates observed at temperatures 0.5 and 1.0, which exceeded the average by 13.94% and 15.6%, respectively. Additionally, in coverage assessments the trend persisted, with higher temperatures producing more effective test cases in terms of decision coverage. Nevertheless, the high standard deviation across temperature settings and programs indicates significant variability, emphasizing that the current model's test generation performance can vary widely depending on code structure and randomness levels.

Regarding mutation scores, the LLM-generated test cases achieved results closer to the baseline, especially with MutPy. While the baseline test cases from Pynguin and the pre-existing tests still slightly outperformed the LLM, the difference was only around 2.03%. This suggests that, despite the challenges of consistently generating valid tests, those tests generated by the LLM that executed successfully were effective in identifying mutants.

Future works could evaluate more advanced generative models, such as newer ChatGPT versions or other LLMs, to improve automated test generation. The scripts from this study can be easily adapted for other models. Also, exploring LLMs' ability to detect specific or equivalent mutants remains open, as current mutation tools do not handle equivalent mutant identification. Moreover, given that the experiment was conducted with a limited number of programs, future studies will explore a larger set of subjects.

A promising direction involves testing varied prompts tailored to specific code characteristics, potentially improving test accuracy and coverage, as this study used a single prompt across all cases for both generation and correction, as well as applying test set minimization for cost efficiency. Combining different LLM models, starting with basic ones and later using more advanced models, also offers opportunities to better meet complex testing requirements and enhance generated test sets.

Finally, we are developing an automated test case generation tool for Python, which follows the strategy presented in this paper for test set generation, fixing, and improvement based on coverage and mutation score.

ARTIFACT AVAILABILITY

Artifacts are available at https://github.com/aurimrv/python_experiments2.

ACKNOWLEDGMENTS

This work is partially supported by Brazilian Funding Agencies FAPESP (Grant nº 2019/23160-0 and 2023/00001-9), CAPES (Grant nº 001 and 88887.888653/2023-00), and CNPq (Grant nº 41137/2021-5).

REFERENCES

[1] J. H. Andrews, L. C. Briand, and Y. Labiche. 2005. Is mutation an appropriate tool for testing experiments?. In *XXVII International Conference on Software Engineering – ICSE'05*. ACM Press, St. Louis, MO, USA, 402–411. https://doi.org/10.1145/1062455.1062530

[2] Filipe Santos Araujo and Auri Marcelo Rizzo Vincenzi. 2020. How far are we from testing a program in a completely automated way, considering the mutation testing criterion at unit level?. In *Anais do Simpósio Brasileiro de Qualidade de Software (SBQS)*. SBC, São Luiz, MA, 151–159. https://doi.org/10.1145/3439961.3439977

[3] Austin Bingham. 2021. Cosmic Ray: mutation testing for Python. (Nov. 2021). https://github.com/sixty-north/cosmic-ray

[4] R. A. DeMillo, R. J. Lipton, and F. G. Sayward. 1978. Hints on Test Data Selection: Help for the Practicing Programmer. *IEEE Computer* 11, 4 (April 1978), 34–43. https://doi.org/10.1109/C-M.1978.218136

[5] Stéphane Ducasse, Manuel Oriol, and Alexandre Bergel. 2011. Challenges to support automated random testing for dynamically typed languages. In *Proceedings of the International Workshop on Smalltalk Technologies (IWST '11)*. Association for Computing Machinery, New York, NY, USA, 1–6. https://doi.org/10.1145/2166929.2166938

[6] Lucca Renato Guerino, Pedro Henrique Kuroishi, Ana Cristina Ramada Paiva, and Auri Marcelo Rizzo Vincenzi. 2024. Static and Dynamic Comparison of Mutation Testing Tools for Python. In *Proceedings of the XXIII Brazilian Symposium on Software Quality (SBQS '24)*. Association for Computing Machinery, New York, NY, USA, 199–209. https://doi.org/10.1145/3701625.3701659

[7] Lucca Renato Guerino and Auri M. R. Vincenzi. 2023. An Experimental Study Evaluating Cost, Adequacy Effectiveness of Pynguin's Test Sets. In *8th Brazilian Symposium on Systematic and Automated Software Testing – SAST'2023*. ACM Press, Campo Grande, MS, 5–14. https://doi.org/10.1145/3624032.3624034

[8] Philipp Hossner, Konrad Hałas, Steven Myint, and Andreas Mueller. 2021. MutPy: a mutation testing tool for Python 3.x source code. (Nov. 2021). https://github.com/mutpy/mutpy

[9] Stephan Lukasczyk. 2019. Generating Tests to Analyse Dynamically-Typed Programs. In *2019 34th IEEE/ACM International Conference on Automated Software Engineering (ASE)*. IEEE, San Diego, CA, USA, 1226–1229. https://doi.org/10.1109/ASE.2019.00146 ISSN: 2643-1572.

[10] Stephan Lukasczyk, Florian Kroiß, and Gordon Fraser. 2023. An empirical study of automated unit test generation for Python. *Empirical Software Engineering* 28, 2 (Jan. 2023), 36. https://doi.org/10.1007/s10664-022-10248-w

[11] Goran Petrović, Marko Ivanković, Gordon Fraser, and René Just. 2021. Does Mutation Testing Improve Testing Practices?. In *Proceedings of the 43rd International Conference on Software Engineering (ICSE '21)*. IEEE Press, Madrid, Spain, 910–921. https://doi.org/10.1109/ICSE43902.2021.00087 Place: Madrid, Spain.

[12] June Sallou, Thomas Durieux, and Annibale Panichella. 2024. Breaking the Silence: the Threats of Using LLMs in Software Engineering. In *Proceedings of the 2024 ACM/IEEE 44th International Conference on Software Engineering: New Ideas and Emerging Results (ICSE-NIER'24)*. Association for Computing Machinery, New York, NY, USA, 102–106. https://doi.org/10.1145/3639476.3639764 event-place: Lisbon, Portugal.

[13] Douglas C. Schmidt. 2025. Software Testing in the Generative AI Era: A Practitioner's Playbook. *Computer* 58, 7 (2025), 147–152. https://doi.org/10.1109/MC.2025.3562940

[14] TIOBE Software BV. 2024. TIOBE Index. https://www.tiobe.com/tiobe-index/

[15] Auri M. R. Vincenzi, Tiago Bachiega, Daniel G. de Oliveira, Simone R. S. de Souza, and José C. Maldonado. 2016. The Complementary Aspect of Automatically and Manually Generated Test Case Sets. In *Proceedings of the 7th International Workshop on Automating Test Case Design, Selection, and Evaluation (A-TEST 2016, Vol. 1)*. ACM, Seattle, USA, 23–30. https://doi.org/10.1145/2994291.2994295 event-place: Seattle, WA, USA.

[16] Sebastian Vogl, Sebastian Schweikl, Gordon Fraser, Andrea Arcuri, Jose Campos, and Annibale Panichella. 2021. EvoSuite at the SBST 2021 Tool Competition. In *2021 IEEE/ACM 14th International Workshop on Search-Based Software Testing (SBST)*. IEEE, Madrid, Spain, 28–29. https://doi.org/10.1109/SBST52555.2021.00012

[17] Junjie Wang, Yuchao Huang, Chunyang Chen, Zhe Liu, Song Wang, and Qing Wang. 2024. Software Testing With Large Language Models: Survey, Landscape, and Vision. *IEEE Trans. Softw. Eng.* 50, 4 (Feb. 2024), 911–936. https://doi.org/10.1109/TSE.2024.3368208 Publisher: IEEE Press.

[18] Quanjun Zhang, Weifeng Sun, Chunrong Fang, Bowen Yu, Hongyan Li, Meng Yan, Jianyi Zhou, and Zhenyu Chen. 2024. Exploring Automated Assertion Generation via Large Language Models. *ACM Trans. Softw. Eng. Methodol.* 34, 13 (Oct. 2024), 25. https://doi.org/10.1145/3699598 Place: New York, NY, USA Publisher: Association for Computing Machinery.